I Script, compilation

Un programme en *Python* est enregistré sous forme d'un *script*.
L'exécution d'un programme, par l'intermédiaire d'un *compilateur* ou *interpréteur*, est l'*interprétation* de son script. Une instruction directement donnée au compilateur sera désignée par :
`>>>` *instruction*

II Indentation et commentaires

L'*indentation* (décalage du texte par rapport à la marge de gauche) est interprétée en langage *Python*. Elle permet de définir des débuts et des fins de séquences (boucles, fonctions, tests...).

Les commentaires suivent le symbole # (*croisillon*, ou *hash* en anglais).
Tout ce qui est écrit sur la ligne après un croisillon n'est pas interprété.

III Symboles opératoires

+	addition	**	puissance
-	soustraction	//	quotient de division euclidienne
*	multiplication	%	reste de division euclidienne
/	division	abs( )	valeur absolue

IV Variables et affectation

Toute variable est définie à l'aide de la commande `=`
On parle de *commande d'affectation*, puisqu'on affecte une valeur (donnée à droite du symbole =) à une variable, qui est alors reconnue par son nom (donné à gauche du =).

Exemples : # *Commentaires*

```
Somme = 33                  # Une variable nommée Somme est créée, et la valeur 33 lui est affectée.
Longueur, Largeur = 10, 6   # Deux variables sont créées. On parle ici d'affectation simultanée.
a = b = c = 5               # Trois variables de même valeur sont crées (affectation multiple)
```

Les variables définies dans le corps principal du script sont les *variables globales*.
Les variables définies dans une séquence (derrière une indentation) sont des *variables locales*. Leur existence n'est garantie que le temps d'exécution de la séquence.

V Commandes de dialogue

• La fonction `print()` permet d'afficher le contenu d'une variable.

```
print(Somme)                # La valeur de la variable Somme s'affiche à l'écran.
```

• La fonction `input()` demande à l'utilisateur une valeur à enregistrer.

```
Prenom = input("Entrer votre prénom")  # Le texte entre guillemets s'affiche, et la suite
                                       # de caractères renseignée par l'utilisateur est enregistrée
                                       # dans la variable nommée Prenom.
```

Remarques : ∗ Si une nouvelle valeur est donnée à une variable pré-existante, cette valeur remplacera l'ancienne valeur (qui est donc perdue).

∗ Pour renseigner avec la commande `input` une valeur numérique, il faudra effectuer une conversion en entier ou en flottant, à l'aide des commandes `int` ou `float` (*voir plus bas*).

VI Les deux principaux types numériques

Python manipule deux sortes de nombres : les nombres entiers relatifs (dans $\mathbf{Z}$), de type `int` (de l'anglais *integer*), et les *flottants*, de type `float`, qui sont ce qu'un ordinateur propose de plus proche d'un réel. Les *flottants* se caractérisent par un . (virgule séparant la partie entière de la partie décimale).

Par exemple, `x = 3` affecte à la variable `x` la valeur 3 de type `int`, tandis que `x = 3.` affecte à la variable `x` la valeur 3 de type `float`.

Les *flottants* sont limités en taille (ils ne dépassent pas $\pm 2^{1024}$, soit environ $\pm 1,8 \times 10^{308}$), et sont définis avec une précision de 10^{-15} : seules les 15 premières décimales sont enregistrées.

Les entiers, en revanche, sont toujours manipulés précisément. Seule la capacité mémoire de l'ordinateur, ou le temps de calcul, peuvent limiter leur manipulation.

On peut convertir un *flottant* en un *entier*, et inversement, en utilisant les commandes : `int` ou `float`.
On peut également manipuler avec *Python* des nombres complexes, de type `complex`.

VII Définir une fonction

Une *fonction* est un algorithme qui utilise des arguments, effectue une série d'instructions et renvoie un résultat. Les *fonctions* sont de type `function`. Pour définir une *fonction*, on respecte la syntaxe suivante :

```
def nomDeLaFonction ( argument1, argument2, ...) :
    ''' une description de la fonction '''
    instruction1
    instruction2
    ...
    return X
```

`X` est typiquement un résultat donné par les instructions, qui utilisent les valeurs `argument1` , `argument2`...

Les arguments sont appelées les *entrées* de la fonction, `X` en est la *sortie* :

$$\text{entrées} \overset{fonction}{\longrightarrow} \text{sortie}$$

La description de la fonction est optionnelle ; elle permet à l'utilisateur d'obtenir de l'aide par la commande : `help(nomDeLaFonction)`

Cette structure doit être respectée, y compris l'indentation, y compris les `:` à la fin de la première ligne. Dès que la commande `return` est exécutée, *Python* considère que la fonction est définie, et stoppe l'exécution.

```
def cube(x) :          # Cette fonction calcule le cube d'un nombre x donné en entrée.
    a = x ** 3         # Une variable provisoire [locale] est créée, égale au cube de x.
    return a           # La fonction renvoie cette variable locale.

def puissance3(x) :    # Une autre façon de procéder.
    return x ** 3      # La fonction renvoie le résultat du calcul de x^3.

def f(x, y) :                      # Cette fonction utilise deux arguments x et y en entrées.
    return 2*x*y + y**2            # ... et renvoie : 2xy + y^2
>>> cube(2)                          8
>>> f(1,4)                           24
```

On peut définir une fonction sans argument. Son appel s'effectue alors de la façon suivante : `>>> f()`

VIII Bibliothèques et modules

Python dispose de *modules* ou de *bibliothèques* (ensembles de modules), contenant de nombreuses commandes, qu'on peut utiliser en les *important*.
Une commande spécifique à un module s'appelle par la syntaxe : `module.commande`.

```
import numpy                  # Importe le module numpy, contenant, entre autres, une commande pi.
>>> numpy.pi                    3.141592653589793
```

Lorsque le nom du module est compliqué, pour éviter d'avoir à l'écrire à chaque fois, on peut renommer le module (pour le temps d'utilisation du programme).

```
import numpy as np            # Importe le module numpy, et le renomme 'np'.
>>> np.pi                       3.141592653589793
```

On peut également importer une simple commande à partir d'un module.

```
from numpy import pi          # Importe la commande pi (et elle seule).
>>> pi                          3.141592653589793
```

On peut enfin importer une commande en la renommant, ou importer toutes les commandes d'un module :

```
from numpy import log as ln   # La commande log qui est le logarithme népérien est importée
                              # et renommée (judicieusement) ln.
>>> ln(10)                      2.302585092994046
from numpy import *           # Toutes les commandes du module numpy sont importées.
>>> cos(pi/4)                   0.7071067811865476
```

Remarque : `import`, `as`, `from` sont des *noms réservés* du langage *Python*.

Fonctions usuelles du module `numpy` :

`cos`, `sin`, `tan`	Les fonctions trigonométriques *cosinus, sinus, tangente* ;
`arccos`, `arcsin`, `arctan`	Les fonctions trigonométriques réciproques des précédentes ;
`sqrt`, `pow`, `exp`	Les fonctions *racine carrée*, *puissance* et *exponentielle* ;
`log`, `log10`, `log2`	Les fonctions *logarithmes népérien*, *décimal* et *de base 2* ;
`floor`, `ceil`, `trunc`	Les fonctions *partie entière*, *partie entière supérieure* et *troncature* ;

IX Le type `Booléen`

Une variable *booléenne* est une variable pouvant prendre 2 valeurs : "vrai" (`True`) ou "faux" (`False`). Les booléens sont utilisés pour effectuer des tests ou poser des conditions.

Opérations sur les booléens : `and`, `or`, `not` permettent d'effectuer les opérations suivantes :

and	True	False
True	True	False
False	False	False

or	True	False
True	True	True
False	True	False

not	
True	False
False	True

Conversion en entiers : les booléens se convertissent en entiers, et réciproquement.

`bool`(0)	renvoie `False`	`int`(`False`)	renvoie 0	`type`(`True`)	renvoie :
`bool`(1)	renvoie `True`	`int`(`True`)	renvoie 1	`<class>, bool`	

Remarque : tout autre nombre que 0, converti en booléen, renvoie `True`.
On peut alors effectuer d'autres opérations sur les *booléens*, qui sont alors automatiquement convertis en entiers. *Exemple* : `>>> True + True` renvoie 2

Le terme *booléen* est donné en l'honneur de *George Boole* (1815-1864).

X Tests, instructions conditionnelles

Un **test** renvoie un booléen. Les commandes de test en *Python* sont :

`>`	*strictement supérieur*	`>=`	*supérieur ou égal*	`==`	*égal*
`<`	*strictement inférieur*	`<=`	*inférieur ou égal*	`!=`	*différent*

Plusieurs types d'objets peuvent être testés (pas seulement des nombres), mais les comparateurs d'ordre ne s'appliquent pas à tous les objets.

```
>>> 3 < 4                          True    # les entiers (int) 3 et 4 sont comparés.
>>> "trois" < "quatre"             False   # les chaînes de caractères (str) sont comparées.
```

Les **instructions conditionnelles** sont : `if` *si...* `elif` *sinon, si ...* `else` *sinon ...*

La syntaxe est la suivante (**a** et **b** désignent des booléens) :

```
if a :
    instructions        # effectuées si et seulement si le booléen a est vrai,
elif b :
    instructions        # effectuées si et seulement si a est faux et b est vrai,
else :
    instructions        # effectuées si et seulement si a et b sont faux.
```

Remarques : ∗ L'indentation, ainsi que les `:` sont nécessaires à la syntaxe.
∗ `elif` et `else` sont optionnels.

XI Boucles

Une *boucle* est une partie du script destinée à être répétée un certain nombre de fois (un nombre fini !).

1 Boucle 'while'

```
while a :
    instructions        # répétées tant que le booléen a est vrai.
```

Si, dans les instructions, rien ne vient modifier le booléen **a**, et si aucune commande spéciale d'arrêt n'est utilisée, alors la boucle peut se répéter indéfiniment (on dira que le programme *boucle*).

Exemple : définir une fonction `f(n)` renvoyant la somme des inverses des entiers k tels que $1 \leqslant ke^k \leqslant n$

```
def f(n) :
    S, k = 0, 1                    # initialisation de la somme S à 0, et de k à 1.
    while k * np.exp(k) <= n :     # Tant que ke^k ⩽ n
        S = S + 1/k                # ajouter 1/k à la somme S
        k = k + 1                  # et passer à l'entier k suivant.
    return S                       # Lorsque la boucle s'achève (ke^k > n), renvoyer la somme S.
```

2 Boucle 'for'

`for` *[variable]* `in` *[objet itérable]* `:`
    instructions # effectuées une fois pour chaque valeur de la variable dans l'objet itérable.

Les objets itérables avec lesquels nous travaillerons sont les objets de type `list` ou `range`, essentiellement.

La commande `range` :

`range(n)` crée l'intervalle entier $[\![0, n-1]\!]$ lorsque $n \in \mathbf{N}^\star$.

`range(n,m)` crée l'intervalle entier $[\![n, m-1]\!]$ lorsque $n < m$ dans $\mathbf{Z}$.

`range(n,m,p)` crée l'ensemble commençant par `n`, finissant **avant** `m`, en augmentant d'un pas de `p`.

Dans les autres cas, ces ensembles sont vides.

Exemples :
`range(6)` crée l'objet itérable $\{0, 1, 2, 3, 4, 5\}$.
`range(3,11)` crée l'objet itérable $\{3, 4, 5, 6, 7, 8, 9, 10\}$.
`range(3,11,2)` crée l'objet itérable $\{3, 5, 7, 9\}$.
`range(11,-3,-4)` crée l'objet itérable $\{11, 7, 3, -1\}$.
`range(11,3,2)` crée l'ensemble vide.

3 Commandes spéciales dans les boucles

• La commande `break` force l'interruption d'une boucle lorsqu'elle est lue.
• La commande `continue` permet d'ignorer les instructions pour une valeur précise de la variable.

Scripts à tester :

```
i = 0
while True :
    print(i)
    i = i + 1
    if i > 100 : break
```

```
for i in range(20) :
    if i % 2 == 0 :
        continue
    print(i)
```

Exemples d'utilisation de boucles :

a Calcul d'une somme

Exemple : que vaut la somme $S = 1 + 3 + 5 + 7 + \cdots + 2025$?

```
S = 0                          # On initialise la somme S à 0.
for k in range(1,2026,2) :     # La variable k prend toutes les valeurs entières de 1 à 2025, de 2 en 2
    S = S + k                  # On ajoute k à la somme S
```

Avec une boucle 'while' :

```
S, x = 0, 1                # On initialise la somme S à 0 et une variable x à 1.
while x < 2026 :           # La boucle s'arrête dès que x ⩾ 2026
    S = S + x              # on ajoute x à S
    x = x + 2              # et on passe à l'entier impair suivant.
```

b Calcul du terme d'une suite récurrente

∗ Soit $(u_n)_{n\in\mathbf{N}}$ la suite définie par : $u_0 = 3$ et pour tout entier naturel n, $u_{n+1} = 1 + \sqrt{u_n}$.
Déterminer une valeur approchée de u_{20}.

```
from numpy import sqrt         # On peut se passer de cette importation en utilisant la puissance 1/2.
u = 3                          # Initialisation du premier terme de la suite.
for _ in range(20) :           # Une boucle répétée 20 fois, la variable n'est pas nommée.
    u = 1 + sqrt(u)            # La nouvelle valeur de u remplace l'ancienne.
```

Exercices

Exercice 1 : Définir en langage *python* les fonctions suivantes :

$(a)\ x \mapsto \dfrac{1}{x^2-1}$ $\qquad$ $(b)\ x \mapsto \dfrac{(x-1)^2}{x^2+2}$ $\qquad$ $(c)\ x \mapsto \ln\left(1+\cos^2(x)\right)$ $\qquad$ $(d)\ (x,y) \mapsto \sqrt{x^2+y^2}$

Exercice 2 : Définir la fonction f telle que : $f(x) = \begin{cases} 2x & \text{si } x \geqslant 2 \\ \ln(x-1)+4 & \text{si } 1 < x < 2 \\ 0 & \text{sinon} \end{cases}$

Exercice 3 : On définit la fonction `mult` par :

```
def mult(a):
    def f(x):
        return x * a
    return f
```

1. Quel est le type de `mult(7)` ?
2. Que vaut `mult(7)(3)` ?

Exercice 4 : Écrire une fonction `Hn` d'argument un entier $n \geqslant 1$ et calculant la somme :

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

Exercice 5 : Même consigne pour les calculs suivants (adapter le nom de la fonction) :

$C_n = 1+4+9+16+\cdots+n^2$ $\qquad$ $P_n = 1+3+3^2+3^3+3^4+\cdots+3^n$

$I_n = \dfrac{1}{1\times 2} + \dfrac{1}{2\times 3} + \dfrac{1}{3\times 4} + \cdots + \dfrac{1}{n\times(n+1)}$ $\qquad$ $T_n = \dfrac{2}{1}\times\dfrac{5}{4}\times\dfrac{10}{9}\times\dfrac{17}{16}\times\cdots\times\dfrac{n^2+1}{n^2}$

Exercice 6

1. Écrire une fonction `factorielle` calculant la factorielle d'un entier $n \in \mathbf{N}^\star$: $n! = 1\times 2\times 3\times\cdots\times n$.
2. Pour n entier entre 1 et 20, afficher la valeur de $n!$
3. En testant votre fonction factorielle, étudier la limite en $+\infty$ de $\dfrac{e^n}{n!}$.
4. Calculer : $S_{20} = \displaystyle\sum_{n=0}^{20} \frac{1}{n!}$. Que peut-on remarquer ?

Exercice 7 : Écrire une fonction `suite` prenant pour argument un entier $n \in \mathbf{N}$ et renvoyant le $n^{\text{ème}}$ terme de la suite $(u_n)_{n\geqslant 0}$ définie par : $u_0 = 10$ et pour tout entier naturel n, $u_{n+1} = \dfrac{1}{2}u_n + 3$.

Exercice 8 : Même consigne pour les suites suivantes :

$$\begin{cases} v_0 = 3 \\ \forall n \in \mathbf{N},\ v_{n+1} = \dfrac{4+v_n}{1+v_n} \end{cases} \qquad \begin{cases} w_0 = 1 \\ \forall n \in \mathbf{N},\ w_{n+1} = -w_n + n + 1 \end{cases}$$

Exercice 9 : On définit la suite de *Fibonacci* $(f_n)_n$ de la façon suivante :

$$f_0 = 0,\ f_1 = 1, \text{ et pour tout } n \geqslant 2,\ f_n = f_{n-1} + f_{n-2}$$

Écrire une fonction prenant pour argument un entier n et renvoyant le $n^{\text{ème}}$ terme de la suite de Fibonacci.

Exercice 10 : Soit $p \in \mathbf{N}^\star$. La suite de *Syracuse* (S^p) est définie de la façon suivante :

$$S_0^p = p \text{ , et pour tout } n \in \mathbf{N},\ S_{n+1}^p = \begin{cases} \dfrac{1}{2}\times S_n^p & \text{si } S_n^p \text{ est pair,} \\ 3\times S_n^p + 1 & \text{sinon.} \end{cases}$$

Exemples : La suite (S^{10}) est $(10, 5, 16, 8, 4, 2, 1, 4, 2, 1, \ldots)$,
la suite (S^7) est $(7, 22, 11, 34, 17, 52, 26, 13, 40, 20, \ldots)$.

1. Écrire une fonction `syracuse` prenant pour arguments un entier $p \in \mathbf{N}^\star$ égal au premier terme d'une suite de Syracuse, et un entier $n \in \mathbf{N}$, et renvoyant les n premiers termes de la suite correspondante.
2. Tester cette fonction avec de nombreuses valeurs de p, en affichant à chaque fois (au moins) les 50 premiers termes de la suite. Que peut-on constater ?
3. Définir le *vol* d'une suite de Syracuse, puis modifier la fonction `syracuse` pour qu'elle renvoie aussi la valeur du vol.
4. Définir l'*altitude* d'une suite de Syracuse, puis modifier la fonction `syracuse` pour qu'elle renvoie aussi la valeur de l'altitude.